Actor Continuation Passing

Efficient and Extensible Request Routing for Event-Driven Architectures

Stefan Plantikow
Zuse Institute Berlin (ZIB)
stefan.plantikow@googlemail.com

Abstract

The logic for handling of application requests to a staged, event-driven architecture is often distributed over different portions of the source code. This complicates changing and understanding the flow of events in the system.

The article presents an approach that extracts request handling logic from regular stage functionality into a set of request scripts. These scripts are executed step-wise by sending continuations that encapsulate their request's current execution state to stages for local processing and optional forwarding of follow-up continuations. A new domain specific language that aims to simplify writing of request scripts is described along with its implementation for the scala actors library. Evaluation results indicate that request handling with actor continuations performs about equally or better compared to using separate stages for request handling logic for scripts of at least 3 sequential steps.

Categories and Subject Descriptors H.2.4 [*Information Systems*]: Systems—Concurrency; D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrency

Keywords Request Routing, Staged Event-Driven Architecture, Continuations, Actor Model, Scala

1. Introduction

Staged, event-driven architectures [4] implement an approach to the design of server software that can provide high degrees of concurrency and throughput. This is achieved by structuring the software as a set of stage that run in separate threads, do not share state, and communicate exclusively via event queues, i.e. follow the actor model of message passing concurrency.

Requests to the server application are enqueued as events at some stage. Handling such an event may involve pure local computation, accessing stage-specific functionality (manipulation of local state and resources), and continuing request handling by sending new events to other stages. This *application logic* can be divided into *stage logic* which must necessarily be executed at a fixed stage and *request logic* which may be executed anywhere as long as it is provided with the required inputs.

The resulting interactions during request handling at runtime can be complex and difficult to understand. Therefore it appears desirable that at least all request logic for a given request type should be implemented in a readable, singular section of the source code. However application logic is typically spread over the implementations of all stages. Additionally, adding new request types may require the introduction of new event types to communicate intermediary values between stages.

This distribution of application logic over different stages reduces the understandability of staged architectures and complicates modifying the handling of application requests. Additionally, it impedes the addition of new request types without changing the source code of existing stages and redeploying parts of the system.

In this article, an approach for extracting request logic into separate source code units that are independent of the implementation of stage logic is presented. The solution is based on sending continuations between stages and CPS-transformation of request handling code. It is unique in that it does neither require additional messages nor leads to source code with deeply nested callbacks. The approach has been implemented as a domain specific languages (DSL) for the scala actors library [1].

2. Intertwined Logic

The distribution of application logic over the system stems from the intertwining of stage-independent global *request logic* and local *stage logic* that actually has to be performed at a specific stage. For a given request, each part of request logic is glued together with some stage logic quite randomly as chosen by the developer. Required intermediary values are sent as part of the event that triggers a block's execution.

Additionally, the result of executing stage logic may determine how and at which concrete stages request handling needs to be continued. This places further burdens on the implementation of intermediary stages, as incoming- and outgoing event types have to be amended with request state, although it might be completely independent from the intended purpose of that stage.

To give an example for this, imagine a simplified system for launching satellites into space. Incoming reqeuests are amended with authentication information in the first stage. In the second stage, this information is then used to authorize the request and eventually launch the rocket. Only after the satellite has begun to operate, some third stage (i.e. the press office) is informed.

First, note how the launching stage needs to know about the overall workflow in order to forward a message to the press office after a succesful launch. Now, imagine that the initial request needs to be amended with extra information (name and owner of satellite) for the press office. Passing this information down requires modifying the events to and from the rocket launching stage with fields for the additional payload, although this extra information is of no importance to actually launching the rocket.

This intertwining of request and stage logic is a case of insufficient separation of concerns, calling for a different way to describe both types of application logic.

3. Separating Request and Stage

Extracting request logic requires a mechanism for interruptible, stateful control-flow. One approach to this is the use of additional coordination stages. In this scheme, request logic is executed by a coordinator stage. Stage logic is executed by sending and receiving events to the executing stage. This solution has some drawbacks: Executing stage logic requires the sending of two messages (request-response), leads to an additional thread context switch back to the coordinator stage (e.g. to access pre-request state), and may necessitate the introduction of new event types to communicate intermediary values. Additionally, special care may be required to avoid overloading of coordinator stages by implementing them in a non-blocking fashion and load-balancing over them.

The *continuation* of a computation at a point in time describes the part of a computation that yet needs to be computed, i.e. describes stack frame state and remaining program. A continuation may be explicitly stored as a value by reifying it as an anonymous function. It may then be restored and executed arbitrary often by calling this function.

This allows to implement pausable, stateful control-flow: Stages receive events that actually are anonymous functions that represent the current continuation of some request. Such incoming continuations are executed by calling them with the executing stage as their sole argument. Thus request continuations gain access to the functionality of local stages. When the execution of a request continuation is about to finish, optionally, the follow-up continuation may be captured in a last step and sent to the next stage where request handling continues.

This *actor continuation passing (ACP)* approach does not require any intermediate stages for the execution of the request logic and thus avoids the introduction of additional messages. It does not require special events for communicating intermediary values since they are contained in the continuation stack frames. On the downside, it requires some overhead for continuation capturing.

4. A Request Handling DSL

Next, Souffleuse, a library for request handling with actor continuation passing is presented. Souffleuse has been implemented in the scala programming language [3] using the scala actor library [1]. Souffleuse provides a *domain specific language (DSL)* for writing request scripts that execute over a set of locally running stages. Request scripts are implemented in terms of a simple set of commands that allow structuring scripts as a sequence of code blocks that are executed at different stages.

- ***v*** ← **remember(*value*)** Bind *value* to *v* for later reuse by the script
- ***v*** ← **compute(*thunk*)** Compute *thunk* at the current local stage. The return value is bound to *v* and may be reused later in the script
- ***s*** ← **goto(*stage*)** Continue request execution at stage *stage* and return a reference *s* for gainining access to stage functionality of *stage* (usually *stage* itself)
- **yield(result)** The yield statement of the **for**-expression may optionally be used to return a result to the initial caller of the script

Routing scripts may be written as **for**-expressions and are executed using two additional primitives of the DSL:

- **run(*forExpr*)** Run *forExpr* and wait until its execution yields a result (blocks current actor)
- **asyncRun(*forExpr*)** Runs *forExpr* without waiting for a result (non-blocking)

As an example, consider the execution of a single remote procedure call:

```
def rpc(targetStage, args) = {
  val request = for(
      stageRep  ← goto(targetStage)
      procResult.  ← compute { stageRep.proc(args) }
  ) yield procResult
  return run(request) }
```

The call is wrapped as a regular function that initially assembles a new request script. The script itself first transfers the execution to the *targetStage* for the RPC using **goto**. Then, the actual RPC is executed at that stage using **compute**, and finally the return value is yielded. To actually execute this request script, it is started with **run**.

Alternatively, request scripts may be written by subclassing the class Play and overriding its **apply** method. This allows to place an upper type bound on all stage instances used by the script.

Stages are implemented by subclassing or instantiating the class Stage (a stock scala actor) and providing it with an exchangeable stage functionality object (called its *Prop*) that is passed to each request script executing at that stage. The Prop instance may be identical with the Stage itself.

5. Implementation

Soeuffleuse implements request handling according to the actor continuation passing approach on top of the scala actor library. Stages are implemented as actors that run in separate threads.[1] Their main loop listens for messages consisting of one-argument anonymous lambda functions. When such a function is received, it is executed and given access to the stage by passing the prop as its first argument. However, explicitely writing out continuation functions can lead to unreadable source code with a nesting level of anonymous lambda functions that is as large as the number of sequentially passed stages.

As a remedy, Souffleuse performs CPS-transform and sending of continuations at stage boundaries. *Continuation Passing Style (CPS)* refers to a control flow graph transformation that replaces regular function return with calling a continuation function passed as an extra argument. Scala's **for**-generator-expressions provide generator objects with continuation functions for the remainer of the for-loop through a CPS-transform done by the scala compiler. How these continuatios are called is left to the generator. This is exploited by Souffleuse's **goto** command to capture the current continuation and send it to a remote stage for execution.

5.1 CPS-Transform in Scala

Next it is shown how Souffleuse exploits Scala's **for**-generator-expressions to implement CPS-transform. Routing scripts are written as expressions of the form:

$$\textbf{for}\ (v_1 \leftarrow e_1, v_2 \leftarrow e_2, \ldots, v_n \leftarrow e_n)\ \textbf{yield}\ r$$

This iterates sequentially from outmost to innermost over the generators e_i. Each v_i is consecutively bound to the values produced by its generator e_i. Results are created by evaluating r in each iteration until e_1 is exhausted.

Scala abstracts from how **for** interprets different types of generators by CPS-transforming the expression and calling abstract

[1] Stages are receive-actors in terms of the scala actor library

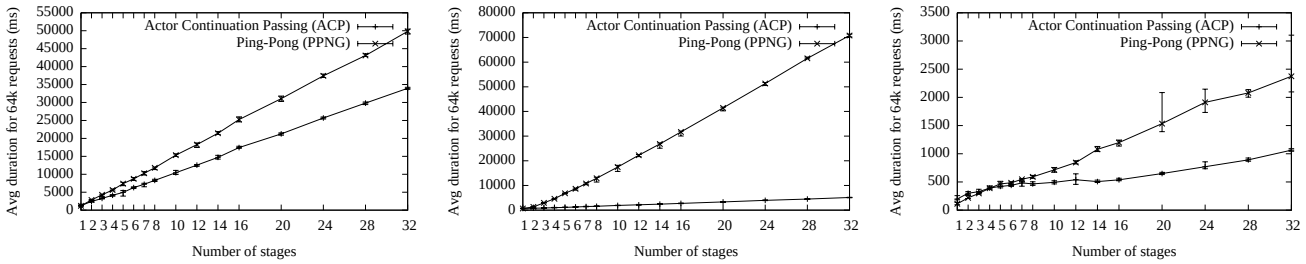

Figure 1. Comparing Actor Continuation Passing against using a coordinator stage when sending messages around a ring

methods on the generators. For example, above expression is transformed by the Scala compiler into:

e_1.flatMap { **case** v_1 $\Rightarrow$
  e_2.flatMap { **case** v_2 $\Rightarrow$ ... e_n.map { r } ... } }

Every { **case** v_i $\Rightarrow$ ... } is an anonymous lambda function that reifies the continuation for the remaining **for**-generator-expression. To make this implicit CPS-transformation usable, the scala standard library contains the abstract class Responder which provides implementations of flatMap and map in terms of a function respond. Respond takes the continuation for the remaining generator-expression as its only argument, generates values, and iterates by calling the continuation with them.

To implement actor continuation passing, Souffleuse associates each stage with a Responder whose respond method simply forwards passed continuations to the stage via message passing:

```
object responder extends Responder[this.type] {
  def respond(k : Actor.this.type ⇒ Unit) : Unit = self.send(k)
}

def asResponder: Responder[Actor.this.type] = responder
```

This mechanism is sufficient to implement the Souffleuse DSL. **goto** returns a responder for its argument as described above. **remember** simply creates a constant responder for a single value. **run** uses the actor library to create a dedicated channel for return values. All other commands are implementable on top of goto and remember.

5.2 Continuation Access

Souffleuse features a type of Stage that allows routing scripts to access the currently running continuation. This may be used to forward continuations to other stages (load balancing), execute the same continuation repeatedly over multiple actors (replication).

5.3 Limitations and future work

The strictly linear nature of generator expressions makes writing routing scripts with non-linear control-flow more difficult and may require the execution of routing sub-scripts. However, even in such a scenario all request routing logic is written in a single routing script.

Souffleuse currently does not yet support exception handling across stage boundaries since the correct specification of such a facility is not obvious to the author at this point, especially considering non-linear request scripts with synchronization.

Beyond exception handling, it would be desirable to extend the library for synchronization in the case of non-linear request scripts. This raises interesting questions in terms of garbage collection and management of auxiliary state by stages that are used as synchronization points.

5.4 Availability

Souffleuse is being made available as open source.

6. Evaluation

Souffleuse has been compared against using coordinator stages in a messages-around-a-ring-of-stages-scenario with different load generation strategies. The results indicate that it performs equally or better when the ring size is ≥ 3 (Fig. 1). With growing ring size, Soufflese converges towards a twofold performance increase since the required number of messages is halved. The use of events consisting of serialized continuation functions appears to have neglible overhead. All tests were conducted on a 8-core machine.

7. Conclusion

The results indicate that actor continuation passing is a viable approach for separating request from stage logic without suffering from the performance penalty introduced by using explicit coordinator stages. Souffleuse implements this approach as a mini-DSL in scala. The implementation eliminates the need for deeply nested callbacks by using the implicit CPS-transformation of scala's **for**-expression. Thus writing request logic as fast and concise request scripts is enabled, while stage logic is implemented separately where it belongs.

Acknowledgments

The author thanks Björn Kolbeck who initially described the problem in the context of XtreemFS [2], a distributed filesystem implemented as a staged architecture.

References

[1] P. Haller and M. Odersky. Actors that unify threads and events. *LNCS*, 4467, Jan 2007.

[2] F. Hupfeld, T. Cortes, B. Kolbeck, E. Focht, M. Hess, J. Malo, J. Marti, J. Stender, and E. Cesario. Xtreemfs - a case for object-based storage in grid data management. In *Proceedings of 33th International Conference on Very Large Data Bases (VLDB) Workshops*, 2007.

[3] M. Odersky, P. Altherr, V. Cremet, B. Emir, and S. Maneth. An overview of the scala programming language. Technical Report LAMP-EPFL 2006-001, EPFL, Jan 2006.

[4] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of SOSP 18*, 2001.